React-Query 101

프론트엔드는 “데이터”가 필요할까?

1ilsang

목차

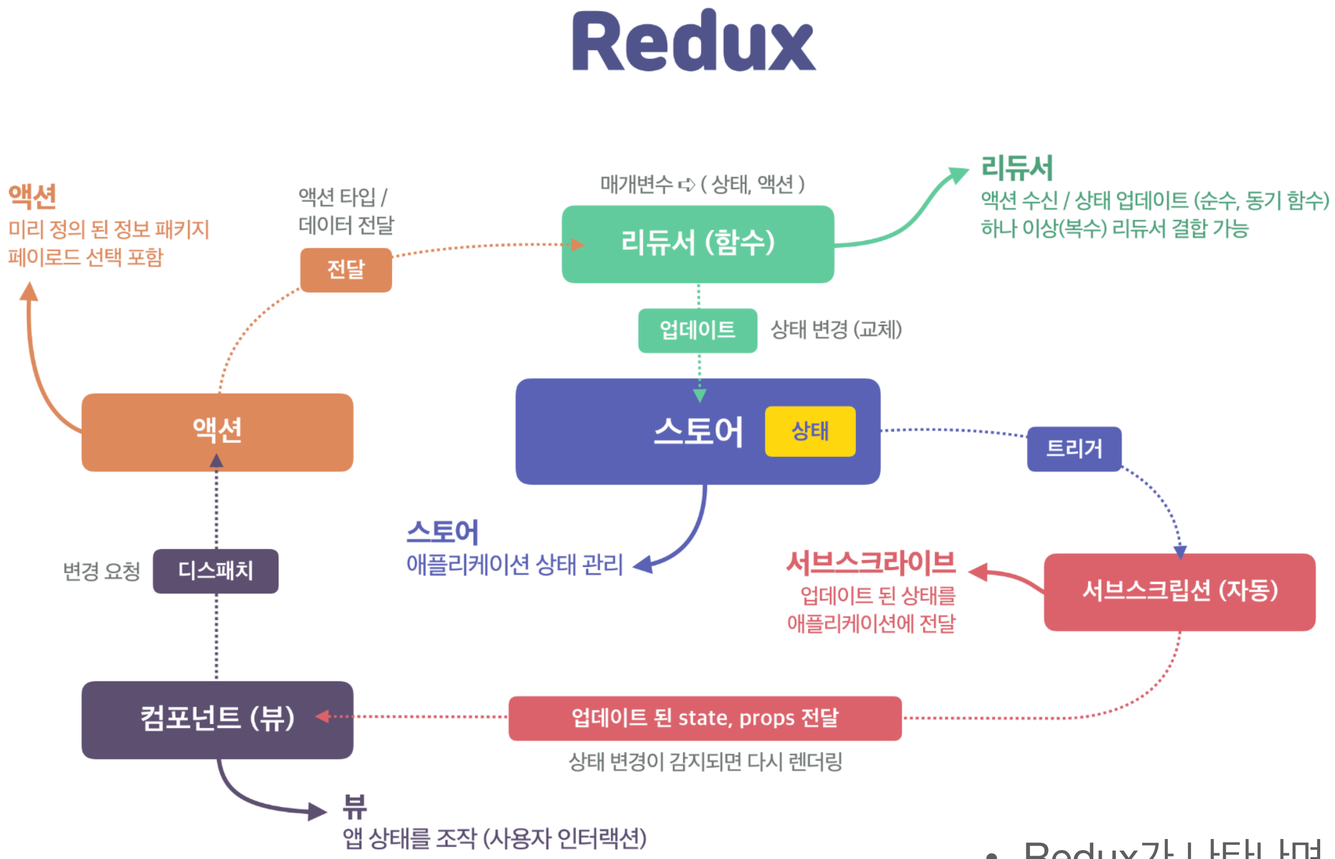
- 리액트 쿼리가 나오기까지의 간단한 역사
- 리액트 쿼리란?
- 간단한 사용법
- 한눈에 보는 리액트 쿼리의 아키텍쳐 및 컴포넌트 플로우
- 마무리

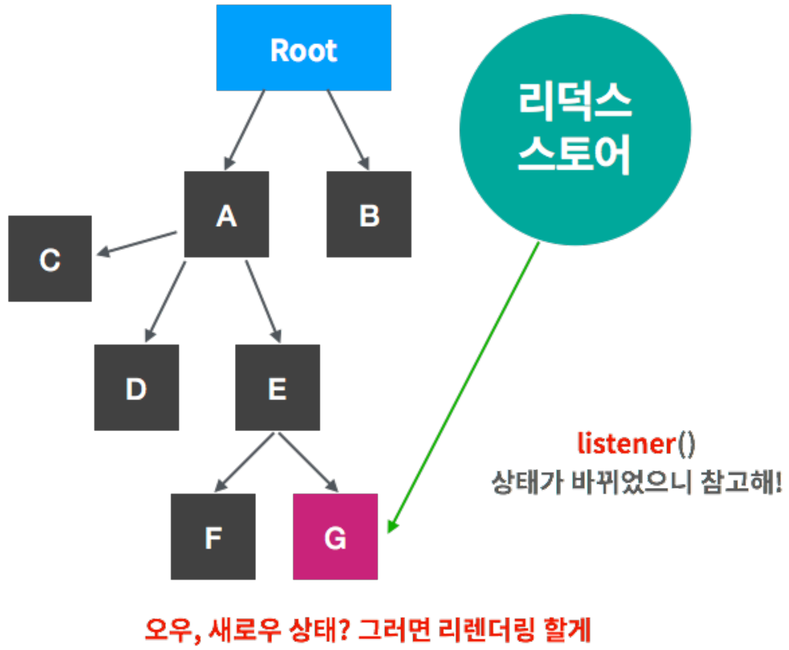
React-Query가 나오기까지...

```
var GlobalObject = {};
```

1. 프론트엔드는 언제나 전역 객체로 인한 어려움이 존재했다.
2. 해당 객체의 값: 상태(state)가 변화(mutation)했다는 것을 알아차리는(observer) 과정을 구성 하기가 어렵다.
3. 유지보수 과정에서 새로운 로직(middleware)을 추가하는 것도 직접 다 구현해야했다.

React-Query가 나오기까지...





- Redux가 나타나며 전역 데이터의 **흐름**에 대한 확실한 개념이 생성됨
- 전역 스토어 덕분에 각 컴포넌트에서 **드릴링 없이 데이터에 쉽게 접근** 할 수 있게 됨

사진 출처
- https://yamoo9.github.io/react-master/lecture/rd-redux.html
- https://velopert.com/3528

React-Query가 나오기까지...

- Redux가 나타나며 전역 데이터의 **흐름**에 대한 확실한 개념이 생성됨
- 전역 스토어 덕분에 각 컴포넌트에서 **드릴링 없이 데이터에 쉽게 접근** 할 수 있게 됨

응 전역에 데이터 다 때려박아~
난 셀렉터만 믿어~

- 유저 아이디
- 게시글 리스트
- 댓글

React-Query가 나오기까지...

- Redux가 나타나며 전역 데이터의 **흐름**에 대한 확실한 개념이 생성됨
- 전역 스토어 덕분에 각 컴포넌트에서 **드릴링 없이 데이터에 쉽게 접근** 할 수 있게 됨

응 전역에 데이터 다 때려박아~
난 셀렉터만 믿어~

서버 데이터와
클라이언트 데이터가 공존하기 시작

- 유저 아이디
- 게시글 리스트
- 댓글

서버 데이터

- Modal on/off
- LocalStorage
- isLoading

클라이언트 데이터

React-Query가 나오기까지...

- Redux가 나타나며 전역 데이터의 **흐름**에 대한 확실한 개념이 생성됨
- 전역 스토어 덕분에 각 컴포넌트에서 **드릴링 없이 데이터에 쉽게 접근** 할 수 있게 됨

응 전역에 데이터 다 때려박아~
난 셀렉터만 믿어~

서버 데이터를
"동기화" 하는 것이 또하나의 과제가 됨.
(Ex. 특정 게시글 상태 변경)

서버 데이터와
클라이언트 데이터가 공존하기 시작

- 유저 아이디
- 게시글 리스트
- 댓글

서버 데이터

- Modal on/off
- LocalStorage
- isLoading

클라이언트 데이터

React-Query가 나오기까지...

- Redux가 나타나며 전역 데이터의 **흐름**에 대한 확실한 개념이 생성됨
- 전역 스토어 덕분에 각 컴포넌트에서 **드릴링 없이 데이터에 쉽게 접근** 할 수 있게 됨

응 전역에 데이터 다 때려박아~
난 셀렉터만 믿어~

서버 데이터를
"동기화" 하는 것이 또하나의 과제가 됨.
(Ex. 특정 게시글 상태 변경)

서버 데이터와
클라이언트 데이터가 공존하기 시작

- 유저 아이디
- 게시글 리스트
- 댓글

서버 데이터

- Modal on/off
- LocalStorage
- isLoading

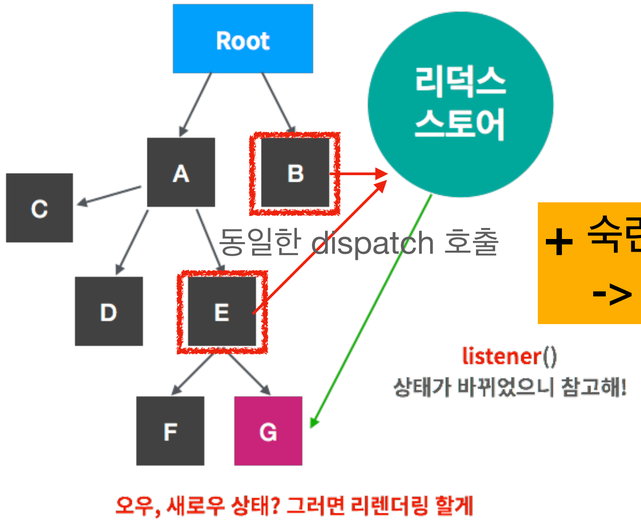
클라이언트 데이터

스토어가 비대해지며
"관심 분리"가 점점 어려워짐

React-Query가 나오기까지...

- Redux가 나타나며 전역 데이터의 **흐름**에 대한 확실한 개념이 생성됨
- 전역 스토어 덕분에 각 컴포넌트에서 **드릴링 없이 데이터에 쉽게 접근** 할 수 있게 됨

응 전역에 데이터 다 때려박아~
난 셀렉터만 믿어~



+ 숙련되지 않은 개발자의 Over-fetch 위험성
-> 왜 동일한 API N번 호출됨..?

서버 데이터를
"동기화" 하는 것이 또하나의 과제가 됨.
(Ex. 특정 게시글 상태 변경)

서버 데이터와
클라이언트 데이터가 공존하기 시작

- 유저 아이디
- 게시글 리스트
- 댓글

서버 데이터

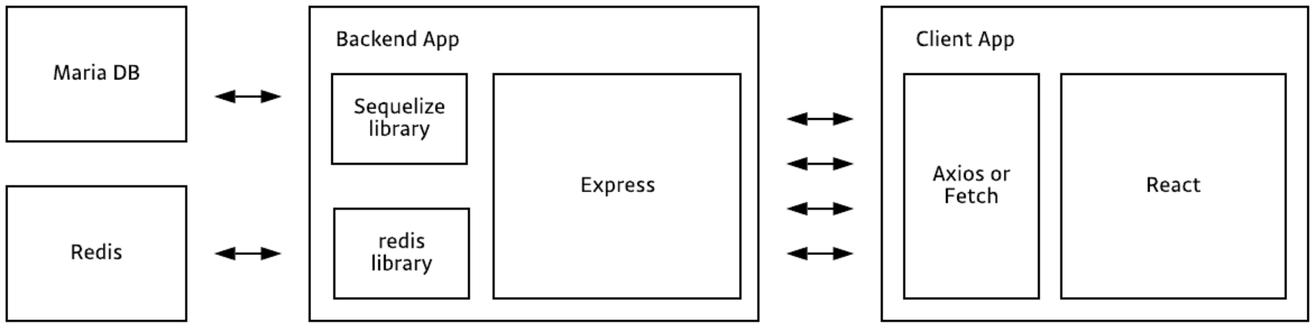
- Modal on/off
- LocalStorage
- isLoading

클라이언트 데이터

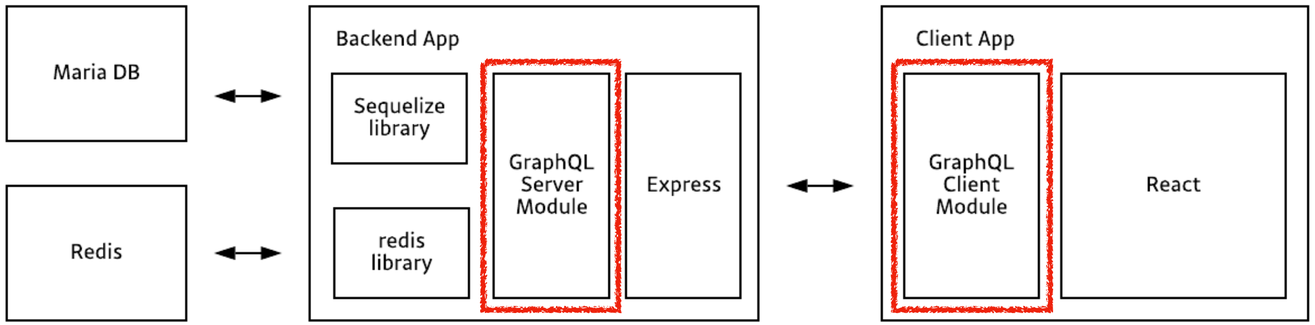
스토어가 비대해지며
"관심 분리"가 점점 어려워짐

React-Query가 나오기까지...

일반 HTTP API 적용 스택



GraphQL 적용 스택



- 아니 매번 **서버 API 조합하는거 짱나**지 않음?
- 어짜피 데이터 사용은 FE에서 하는데 **우리가 달란대로 줘**
- **그렇게 되면 FE에서 데이터를 조합/저장 할 필요가 없네?**

사진 출처

- https://tech.kakao.com/2019/08/01/graphql-basic/

React-Query가 나오기까지...


```
const GET_DOG_PHOTO = gql`
  query Dog($breed: String!) {
    dog(breed: $breed) {
      id
      displayImage
    }
  }
`;

function DogPhoto({ breed }) {
  const { loading, error, data } = useQuery(GET_DOG_PHOTO, {
    variables: { breed },
  });

  if (loading) return null;
  if (error) return `Error! ${error}`;

  return (
    <img src={data.dog.displayImage} style={{ height: 100, width: 100
  );
}
```

- 아니 매번 **서버 API 조합하는거 짱나**지 않음?
- 어짜피 데이터 사용은 FE에서 하는데 **우리가 달란대로 줘**

- Apollo Client 에서 제공하는 **useQuery가 매우 편리**함!
- **캐싱을 지원**해 Over-fetch 문제가 없어지게 된다.
- **!!? 서버 데이터를 관리할 필요가 없네?**

사진 출처
- https://www.apollographql.com/docs/react/data/queries/

React-Query가 나오기까지...


```
const GET_DOG_PHOTO = gql`
  query Dog($breed: String!) {
    dog(breed: $breed) {
      id
      displayImage
    }
  }
`;

function DogPhoto({ breed }) {
  const { loading, error, data } = useQuery(GET_DOG_PHOTO, {
    variables: { breed },
  });

  if (loading) return null;
  if (error) return `Error! ${error}`;

  return (
    <img src={data.dog.displayImage} style={{ height: 100, width: 100
  );
}
```

- 아니 매번 **서버 API 조합하는거 짱나**지 않음?
- 어짜피 데이터 사용은 FE에서 하는데 **우리가 달란대로 줘**

하지만 서버의 공수가 크기 때문에 실제로 사용 적용 되기까지 어려움이 있었음.

- Apollo Client 에서 제공하는 **useQuery가 매우 편리**함!
- **캐싱을 지원**해 Over-fetch 문제가 없어지게 된다.
- **!!? 서버 데이터를 관리할 필요가 없네?**

React-Query가 나오기까지...


```
const GET_DOG_PHOTO = gql`
  query Dog($breed: String!) {
    dog(breed: $breed) {
      id
      displayImage
    }
  }
`;

function DogPhoto({ breed }) {
  const { loading, error, data } = useQuery(GET_DOG_PHOTO, {
    variables: { breed },
  });

  if (loading) return null;
  if (error) return `Error! ${error}`;

  return (
    <img src={data.dog.displayImage} style={{ height: 100, width: 100
  );
}
```

REST API 에서 Apollo의 강점을 가질수 있다면 어떨까?
여기부터 React-Query의 시작!

- 아니 매번 **서버 API 조합하는거 짱나**지 않음?
- 어짜피 데이터 사용은 FE에서 하는데 **우리가 달란대로 줘**
- Apollo Client 에서 제공하는 **useQuery가 매우 편리**함!
- **캐싱을 지원**해 Over-fetch 문제가 없어지게 된다.
- **!!? 서버 데이터를 관리할 필요가 없네?**

React Query 란?

Performant and powerful data synchronization for React

Fetch, cache and update data in your React and React Native applications all without touching any "global state".

리액트를 위한 고성능의 강력한 데이터 동기화

“전역 상태”를 건드리지 않고 React 앱의 데이터를 가져오고 캐시하며 업데이트 합니다.

- 리액트 쿼리는 GraphQL의 강점을 REST API로 가져온 것과 같다.
- SWR 적용 및 Promise 반환으로 Loading, error handling 등을 기본적으로 지원한다.

사진 출처
- https://react-query-v3.tanstack.com/

SWR?

5. Cache-Control 확장기능 : stale-while-revalidate

1. **Cache-Control: max-age=1, stale-while-revalidate=59**

- HTTP 요청이 1초 내에 반복적으로 발생할 경우, 캐시된 값을 별도의 검증없이 그대로 반환한다.
- HTTP 요청이 1 ~ 60 초 사이에 반복적으로 발생할 경우, 캐시된 값은 이미 낡았지만(out-of-date) 캐싱된 값을 반환한다. 이와 동시에 백그라운드에서 향후 사용을 위해 캐시를 새로운 값으로 채우도록 재검증 요청이 이루어진다.
- 60초 이후에 들어온 HTTP 요청에 대해선 다시 서버에 요청을 보낸다.

Age of previously cached response at time of next request:

0-1 seconds	1-60 seconds	60+ seconds
Cached response is fresh, and used as-is to fulfill browser request. No revalidation.	Cached response is stale, but used as-is to fulfill browser request. Network response used to revalidate in the background.	Cached response is stale, and is not used at all. Network response used to fulfill the browser request, and to populate the cache.

사진 출처
- https://jooonho.com/web/2021-05-16-stale-while-revalidate

간단한 사용법

이후의 내용은 모두 React-Query v4.3.9를 기준으로 함.

```tsx
const queryClient = new QueryClient();

const App: FunctionComponent = () => {
  return (
    <QueryClientProvider client={queryClient}>
      <Body />
      <ReactQueryDevtools initialIsOpen={false} />
    </QueryClientProvider>
  );
};

export default App;
```

```tsx
interface ServerResponse {
  result: string;
}

interface CustomError {
  message: string;
}

type InitResult = string;
```

```tsx
const Body: FunctionComponent = () => {
  const { isLoading, error, data } = useQuery<
    ServerResponse,
    CustomError,
    InitResult
  >(["init"], () => fetch("http://localhost:3030").then((res) => res.json()), {
    select: (res) => {
      return res.result;
    },
    onSuccess: () => {
      console.log("Good!");
    },
  });
  if (isLoading) return <h1>Loading...</h1>;
  if (error) return <h1>An error has occurred: {error.message}</h1>;

  return (
    <div>
      <h1>{data}</h1>
    </div>
  );
};
```

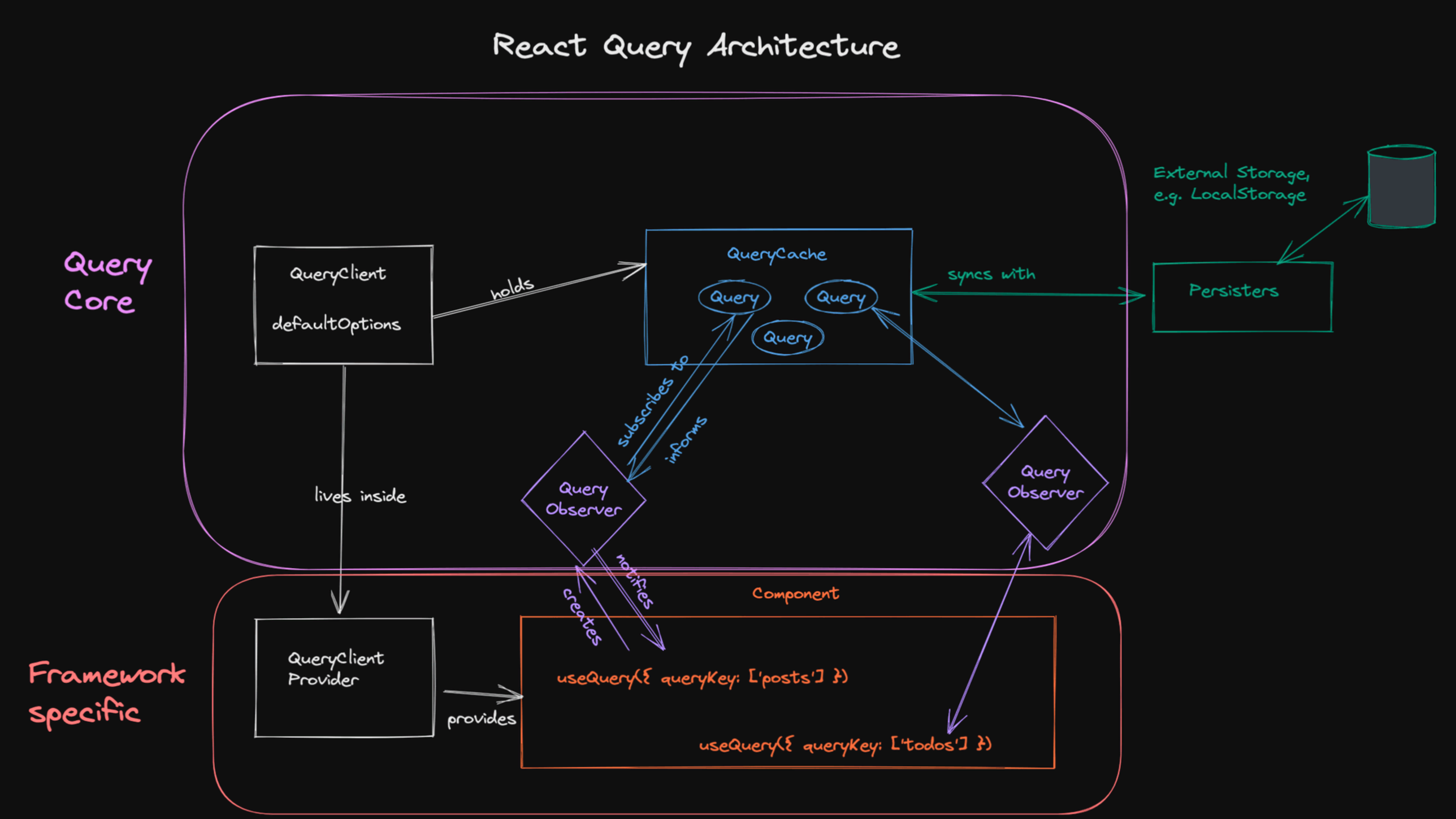
Github
- https://github.com/1ilsang/react-query-practice

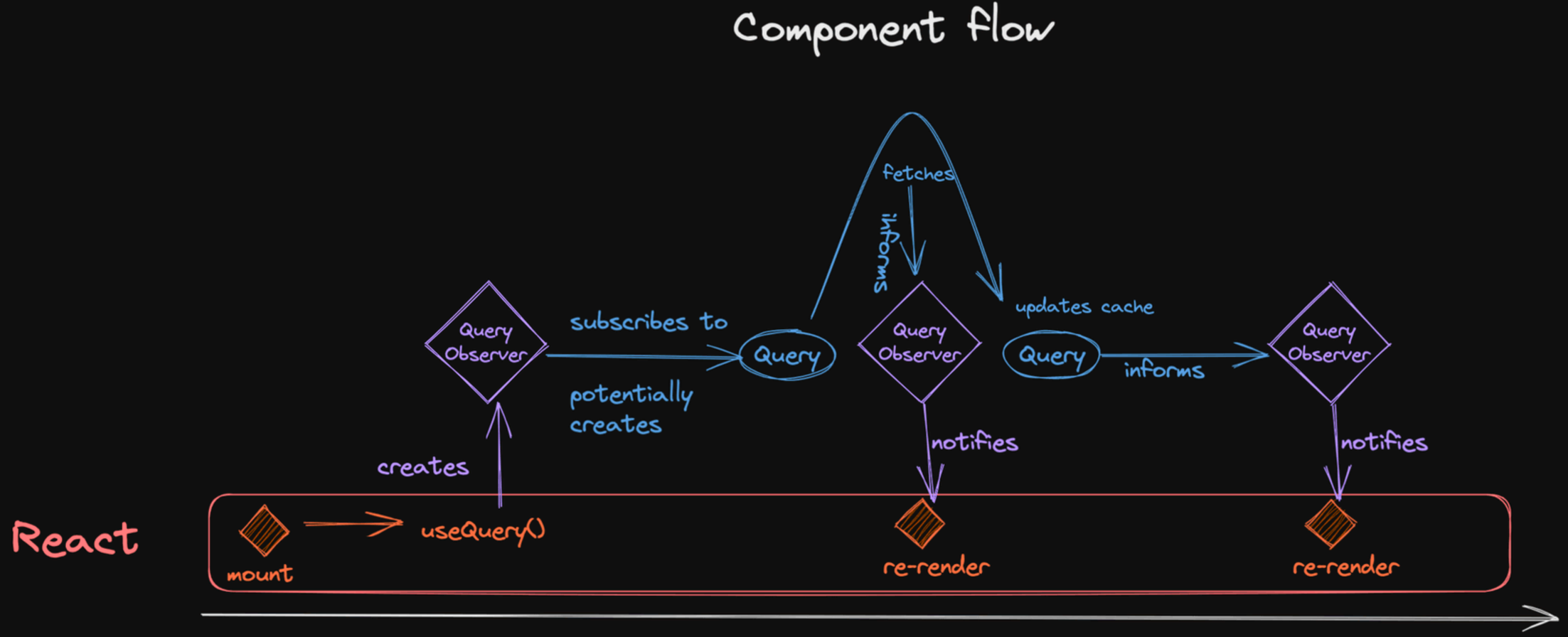
간단한 사용법

1. 전체적인 코드 설명
2. 네트워크탭 확인(다중 호출?)
3. staleTime 설명
4. Suspense 적용
5. refetch
6. isFetch vs isLoading
    1. Flash component
7. QueryKey
    1. 동일 id
8. initialData vs placeholderData
    1. List component
9. Mutation
    1. useList
10. invalidateQueries vs setQueryData

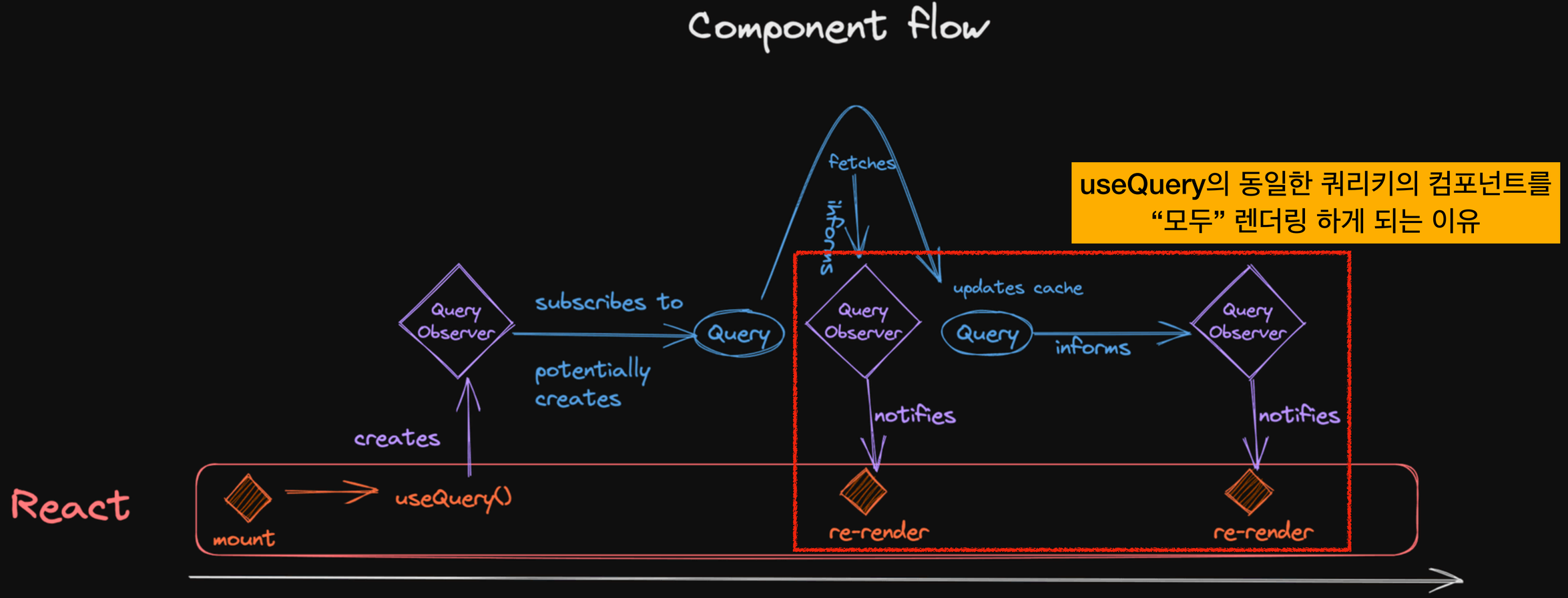
Github
- https://github.com/1ilsang/react-query-practice

React Query Architecture





- 컴포넌트가 마운트되면 `Observer`를 생성하는 `useQuery`를 호출합니다.
- `Observer`는 `QueryCache`에 있는 `Query`를 구독합니다.
- 해당 구독은 `Query` 생성을 트리거하거나(아직 존재하지 않는 경우) 데이터가 오래된 것으로 간주되는 경우 백그라운드 fetching을 트리거할 수 있습니다.
- fetching을 시작하면 `Query`의 상태가 변경되므로 `Observer`에게 이에 대한 정보가 제공됩니다.
- 그런 다음 `Observer`는 몇 가지 최적화를 실행하고 잠재적으로 업데이트에 대해 컴포넌트에 알리면 새 상태를 렌더링할 수 있습니다.
- `Query` 실행이 끝나면 `Observer`에게도 이 사실을 알립니다.



- 컴포넌트가 마운트되면 `Observer`를 생성하는 `useQuery`를 호출합니다.
- `Observer`는 `QueryCache`에 있는 `Query`를 구독합니다.
- 해당 구독은 `Query` 생성을 트리거하거나(아직 존재하지 않는 경우) 데이터가 오래된 것으로 간주되는 경우 백그라운드 fetching을 트리거할 수 있습니다.
- fetching을 시작하면 `Query`의 상태가 변경되므로 `Observer`에게 이에 대한 정보가 제공됩니다.
- 그런 다음 `Observer`는 몇 가지 최적화를 실행하고 잠재적으로 업데이트에 대해 컴포넌트에 알리면 새 상태를 렌더링할 수 있습니다.
- `Query` 실행이 끝나면 `Observer`에게도 이 사실을 알립니다.

마무리

- + useInifiniteQuery
  - 무한 스크롤 구성할때 매우 편리하다.
- 서버 데이터를 저장하지 말자

:wq!